

## 1. Difference between \$\* and \$@ [Extra Options also]

There are special parameters that allow accessing all the command-line arguments at once. \$\* and \$@ both will act the same unless they are enclosed in double quotes, "".

Both the parameters specify the command-line arguments. However, the "\$\*" special parameter takes the entire list as one argument with spaces between and the "\$@" special parameter takes the entire list and separates it into separate arguments.

### Extra Options:

\$0

The filename of the current script.

\$n

These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is \$1, the second argument is \$2, and so on).

\$#

The number of arguments supplied to a script.

\$\*

All the arguments are double quoted. If a script receives two arguments, \$\* is equivalent to \$1 \$2.

\$@

All the arguments are individually double quoted. If a script receives two arguments, \$@ is equivalent to \$1 \$2.

\$?

The exit status of the last command executed.

\$\$

The process number of the current shell. For shell scripts, this is the process ID under which they are executing.

\$!

The process number of the last background command.

## 2. What do you mean by Command Substitution.

Command Substitution:

Command substitution is a mechanism that is followed by programmers in a shell script. In this mechanism, the output of a command replaces the command itself. Shell

operates the expansion by executing a command and then replacing the command substitution with the standard output of the command. In simple words, the output of a UNIX command is bundled and then used as a command.

To understand it in a better way, let us consider an example. The seq command in Linux is used to print numbers from START to END in steps of INCREMENT.

Syntax:

```
seq START INCREMENT END
```

Return type:

Prints numbers from START to END each in the new line by the difference of INCREMENT.

Example:

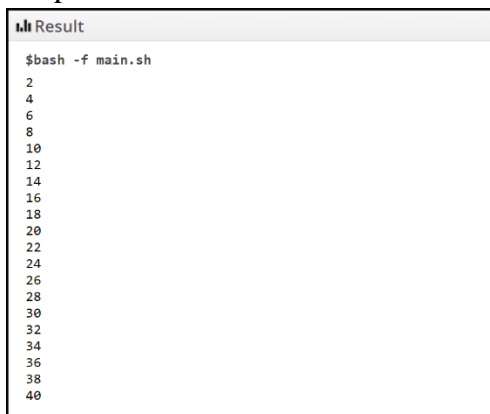
In the below script we are printing numbers from 2 to 40 with a difference of 2. In other words, we are printing even numbers up to 40.

```
#!/bin/sh
```

```
# your code goes here
```

```
seq 2 2 40
```

Output:

A terminal window titled "Result" showing the execution of a script. The prompt is "\$bash -f main.sh". The output consists of even numbers from 2 to 40, listed one per line: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40.

```
Result
$bash -f main.sh
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
```

### 3. Define input output Redirection.

Redirection can be defined as changing the way from where commands read input to where commands sends output. You can redirect input and output of a command.

For redirection, meta characters are used. Redirection can be into a **file** (shell meta characters are angle **brackets** '<', '>') or a **program** ( shell meta characters are **pipesymbol** '|').

---

## Standard Streams In I/O Redirection

The bash shell has three standard streams in I/O redirection:

- **standard input (stdin)** : The stdin stream is numbered as stdin (0). The bash shell takes input from stdin. By default, keyboard is used as input.
  - **standard output (stdout)** : The stdout stream is numbered as stdout (1). The bash shell sends output to stdout. Output goes to display.
  - **standard error (stderr)** : The stderr stream is numbered as stderr (2). The bash shell sends error message to stderr. Error message goes to display.
- 

## Redirection Into A File

Each stream uses redirection commands. Single bracket '>' or double bracket '>>' can be used to redirect standard output. If the target file doesn't exist, a new file with the same name will be created.

## Overwrite

Commands with a single bracket '>' **overwrite** existing file content.

- > : standard output
- < : standard input
- 2> : standard error

Note: Writing '1>' or '>' and '0<' or '<' is same thing. But for stderr you have to write '2>'.

## Syntax:

1. cat > <fileName>

## Example:

1. cat > sample.txt

```
sssit@JavaTpoint: ~  
sssit@JavaTpoint:~$ cat > sample.txt  
a  
b  
c  
sssit@JavaTpoint:~$ cat sample.txt  
a  
b  
c  
sssit@JavaTpoint:~$ cat > sample.txt  
d  
e  
f  
sssit@JavaTpoint:~$ cat sample.txt  
d  
e  
f  
sssit@JavaTpoint:~$
```

Look at the above snapshot, command "cat > sample.txt" has created 'sample.txt' with content 'a, b, c'. Same file 'sample.txt' is created again with command "**cat > sample.txt**" and this time it overwrites earlier file content and only displays 'd, e, f'.

---

## Append

Commands with a double bracket '>>' **do not overwrite** the existing file content.

- >> - standard output
- << - standard input
- 2>> - standard error

### Syntax:

1. cat >> <fileName>

### Example:

1. cat >> sample.txt

```
sssit@JavaTpoint: ~  
sssit@JavaTpoint:~$ cat >> sample.txt  
a  
b  
c  
sssit@JavaTpoint:~$ cat sample.txt  
a  
b  
c  
sssit@JavaTpoint:~$ cat >> sample.txt  
d  
e  
f  
sssit@JavaTpoint:~$ cat sample.txt  
a  
b  
c  
d  
e  
f  
sssit@JavaTpoint:~$
```

Look at the above snapshot, here again we have created two files with the same name using '>>' in command "**cat >> sample.txt**". But this time, content doesn't overwrite and everything is displayed.

---

### Redirection Into A Program

Pipe redirects a stream from one **program** to another. When pipe is used to send standard output of one program to another program, first program's data will not be displayed on the terminal, only the second program's data will be displayed.

Although the functionality of pipe may look similar to that of '>' and '>>' but has a significance difference. Pipe redirects data from one program to another while brackets are only used in redirection of files.

### Example:

1. `ls *.txt | cat > txtFile`

```
sssit@JavaTpoint: ~  
sssit@JavaTpoint:~$ ls *.txt | cat > txtFile  
sssit@JavaTpoint:~$  
sssit@JavaTpoint:~$ cat txtFile  
dupli.txt  
format.txt  
marks.txt  
msg.txt  
sssit@JavaTpoint:~$
```

Look at the above snapshot, command `"ls *.txt | cat > txtFile"` has put all the '.txt' files into a newly created file 'txtFile'.

#### **4. What is the Purpose of Superblock.**

In Unix-like operating systems, the superblock plays a crucial role in managing the file system. The superblock is a data structure that contains important metadata and information about the file system. Its primary purpose is to keep track of the file system's overall structure and state. Here are some key functions and information contained in the superblock:

1. **File System Information:** The superblock contains essential details about the file system, such as its type (e.g., ext4, XFS, etc.), size, block size, and the number of inodes (data structures that represent files and directories).
2. **File System Integrity:** It stores information that helps maintain the integrity of the file system, such as a checksum to detect and correct errors in the file system structure.
3. **Block Allocation:** The superblock tracks the allocation of data blocks within the file system. It keeps information about which blocks are free and which are in use.
4. **Mount Point Information:** The superblock holds data related to the file system's mount point, allowing the system to locate and mount the file system at the appropriate directory.
5. **Inode Information:** It keeps details about inodes, including the size and location of the inode table, which is essential for managing files and directories.
6. **Journaling Information:** If the file system supports journaling (a technique for maintaining file system consistency in case of crashes or power failures), the superblock contains information about the journal.

The superblock is typically located at a fixed position on the disk, and it's one of the first data structures read when the file system is mounted. It's essential for the proper functioning and management of the file system, as it provides the operating system with the necessary information to access and manipulate files and directories.

#### **5. Explain Sort Command in Unix.**

SORT command is used to sort a file, arranging the records in a particular order. By default, the sort command sorts file assuming the contents are ASCII. Using options in the sort command can also be used to sort numerically.

- SORT command sorts the contents of a text file, line by line.
- sort is a standard command-line program that prints the lines of its input or concatenation of all files listed in its argument list in sorted order.
- The sort command is a command-line utility for sorting lines of text files. It supports sorting alphabetically, in reverse order, by number, by month, and can also remove duplicates.

- The sort command can also sort by items not at the beginning of the line, ignore case sensitivity, and return whether a file is sorted or not. Sorting is done based on one or more sort keys extracted from each line of input.
- By default, the entire input is taken as the sort key. Blank space is the default field separator.

**The sort command follows these features as stated below:**

1. Lines starting with a number will appear before lines starting with a letter.
2. Lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet.
3. Lines starting with an uppercase letter will appear before lines starting with the same letter in lowercase.

### **Examples**

Suppose you create a data file with name file.txt:

Command :

```
$ cat > file.txt
```

abhishek

chitransh

satish

rajan

naveen

divyam

harsh

**Sorting a file: Now use the sort command**

**Syntax :**

```
$ sort filename.txt
```

Command:

```
$ sort file.txt
```

Output :

abhishek

chitransh

divyam

harsh

naveen

rajan

satish

**Note:** This command does not actually change the input file, i.e. file.txt.

**Sort function with mix file i.e. uppercase and lower case:** When we have a mix file with both uppercase and lowercase letters then first the upper case letters would be sorted following with the lower case letters.

**Example:**

Create a file mix.txt

Command :

```
$ cat > mix.txt
```

abc

apple

BALL

Abc

bat

Now use the sort command

Command :

```
$ sort mix.txt
```

Output :

Abc

BALL

abc

apple

bat

**Options with sort function:**

**1. -o Option:** Unix also provides us with special facilities like if you want to write the **output to a new file**, output.txt, redirects the output like this or you can also use the built-in sort option -o, which allows you to specify an output file.

Using the -o option is functionally the same as redirecting the output to a file.

**Note:** Neither one has an advantage over the other.

**Example:** The input file is the same as mentioned above.

**Syntax:**

```
$ sort inputfile.txt > filename.txt
```

```
$ sort -o filename.txt inputfile.txt
```

Command:

```
$ sort file.txt > output.txt
```

```
$ sort -o output.txt file.txt
```

```
$ cat output.txt
```



Output :

abhishek  
chitransh  
divyam  
harsh  
naveen  
rajan  
satish

**2. -r Option: Sorting In Reverse Order:** You can perform a reverse-order sort using the -r flag. the -r flag is an option of the sort command which sorts the input file in reverse order i.e. descending order by default.

**Example:** The input file is the same as mentioned above.

**Syntax :**

\$ sort -r inputfile.txt

Command :

\$ sort -r file.txt

Output :

satish  
rajan  
naveen  
harsh  
divyam  
chitransh  
abhishek

**3. -n Option:** To sort a file **numerically** used -n option. -n option is also predefined in Unix as the above options are. This option is used to sort the file with numeric data present inside.

**Example :**

Let us consider a file with numbers:

Command :

\$ cat > file1.txt

50  
39  
15  
89  
200

**Syntax:**

```
$ sort -n filename.txt
```

Command :

```
$ sort -n file1.txt
```

Output :

15

39

50

89

200

**4. -nr option:** To sort a file with **numeric data in reverse order** we can use the combination of two options as stated below.

**Example:** The numeric file is the same as above.

**Syntax :**

```
$ sort -nr filename.txt
```

Command :

```
$ sort -nr file1.txt
```

Output :

200

89

50

39

15

**5. -k Option:** Unix provides the feature of sorting a table on the **basis of any column number by using -k option.**

Use the -k option to sort on a certain column. For example, use “-k 2” to sort on the second column.

**Example :**

Let us create a table with 2 columns

```
$ cat > employee.txt
```

manager 5000

clerk 4000

employee 6000

peon 4500

director 9000

guard 3000

**Syntax :**

```
$ sort -k filename.txt
```

Command :

```
$ sort -k 2n employee.txt
```

```
guard 3000
```

```
clerk 4000
```

```
peon 4500
```

```
manager 5000
```

```
employee 6000
```

```
director 9000
```

**6. -c option:** This option is used to check if the **file given is already sorted or not** & checks if a file is already sorted pass the -c option to sort. This will write to standard output if there are lines that are out of order. The sort tool can be used to understand if this file is sorted and which lines are out of order

**Example :**

Suppose a file exists with a list of cars called cars.txt.

Audi

Cadillac

BMW

Dodge

**Syntax :**

```
$ sort -c filename.txt
```

Command :

```
$ sort -c cars.txt
```

Output :

```
sort: cars.txt:3: disorder: BMW
```

**Note : If there is no output then the file is considered to be already sorted**

**7. -u option:** To **sort and remove duplicates** pass the -u option to sort. This will write a sorted list to standard output and remove duplicates.

This option is helpful as the duplicates being removed give us a redundant file.

**Example:** Suppose a file exists with a list of cars called cars.txt.

Audi

BMW

Cadillac

BMW

Dodge

**Syntax :**

```
$ sort -u filename.txt
```

Command :

```
$ sort -u cars.txt
```

```
$ cat cars.txt
```

Output :

Audi

BMW

Cadillac

Dodge

**8. -M Option:** To **sort by month** pass the -M option to sort. This will write a sorted list to standard output ordered by month name.

**Example:**

Suppose the following file exists and is saved as months.txt

```
$ cat > months.txt
```

February

January

March

August

September

**Syntax :**

```
$ sort -M filename.txt
```

Using The -M option with sort allows us to order this file.

Command :

```
$ sort -M months.txt
```

```
$ cat months.txt
```

Output :

January

February

March

August

September

## 6. Explain Chmod 615 to any file.

Chmod (change mode) is a UNIX/Linux command that you can use to change permissions on a file. Here we will explain how the chmod 615 UNIX/Linux command changes your file permissions. We start by color-coding the three digits in 615, like this, so it is easy to follow along:

615

The three digits refer to three different permissions:

6: Owner Permissions  
1: Group Permissions  
5: Everyone Permissions

Furthermore, chmod digits give the following kind of permissions:

1 = Execute  
2 = Write  
3 = Write & Execute  
4 = Read  
5 = Read & Execute  
6 = Read & Write  
7 = Read, Write & Execute

Now, when we put it all together, we can see what the chmod 615 command means and what it will do to your file permissions:

6: Owner can read and write  
1: Group can execute  
5: Everyone can read and execute

## 7. umask and all permission to files

What is UMASK.?

UMASK is a shell built-in feature which controls the default permissions of an object that you create. When you create an object (either a file or a folder) the operating system checks your UMASK and based upon that UMASK OS assigns permission for that object. When you are dealing with UMASK you should keep two things in mind, the full permission of an object is 777 and we cannot set execution permission for a file using UMASK.

How UMASK works.?

Subtract the umask value from 777, result will be the permission of the directory and if the result contains any execution permission, exclude the execution bit and it will be the permission of the file. Note, however, that files are not usually created with the execute permission by default, so the final permissions for files will omit the “x” permission.

How the calculation should be done

For example your UMASK is 222

```
777 - Full permission
222  UMASK value
-----
555 => This will be the permission of the Directory.
555 => r-x,r-x,r-x
```

The result contains execution permission. So exclude x bit from the result, it will be the permission of the file. r--,r--,r-- => 444

Final result -: If UMASK is 222 then directory's permission will be 555 and file's permission will be 444.

For example your UMASK is 333

```
777 - Full Permission
333  UMASK value
-----
444 => This will be the permission of the Directory.
444 => r--,r--,r--
```

The result contains no execution permission. So no need to exclude x Bit.

Final result -: If UMASK is 333 then directory's permission will be 444 and file's permission will be 444.

For example your UMASK is 666

```
777 - Full Permission
666  UMASK value
-----
111 => This will be the permission of the Directory.
111 => --x,--x,--x
```

The result contains only execution permission. So if we exclude them there will be nothing left for the file permission and it will be blank.

Final result -: If UMASK is 666 then directory's permission will be 111 and file's permission will be 000.

For example your UMASK is 022

```
777 - Full Permission
022  Umask value
-----
755 => This will be the permission of the Directory.
755 => rwxr-xr-x
```

The result contains execution permission. So exclude x bit from the result, it will be the permission of the file. rw-r--r-- => 644

Final result -: If UMASK is 022 then directory's permission will be 755 and file's permission will be 644.

For example your UMASK is 002

777 - Full Permission  
002 Umask value

-----

775 => This will be the permission of the Directory.

775 => rwx,rwx,r-x

The result contains execution permission. So exclude x bit from the result, it will be the permission of the file. rw-rw-r-- => 664

Final result -: If UMASK is 002 then directory's permission will be 775 and file's permission will be 664.

## 8. List out Functions of Kernal

The kernel in Unix-like operating systems serves as the core component that manages system resources and provides an interface for user-level processes to interact with the hardware.

Here is a list of some of the key functions of the kernel in Unix:

1. **Process Management:** The kernel creates, schedules, and terminates processes, ensuring efficient use of CPU time and managing process communication.
2. **Memory Management:** It allocates and manages system memory, including virtual memory, paging, and swapping to and from secondary storage.
3. **File System Management:** The kernel manages file I/O, file access permissions, and maintains the file system structure.
4. **Device Management:** It handles device drivers, enabling communication between hardware devices and user-level applications.
5. **Interprocess Communication (IPC):** The kernel provides mechanisms for processes to communicate and share data, such as pipes, sockets, and message queues.
6. **System Call Interface:** The kernel offers a set of system calls that user-level programs can invoke to request services, like file operations, process control, and network communication.
7. **Network Stack:** It manages network communication, including protocol handling, socket management, and routing.
8. **Interrupt Handling:** The kernel responds to hardware interrupts, ensuring devices are serviced promptly.

9. **Security:** It enforces access controls and security policies, protecting the system and user data.
10. **Scheduling:** The kernel schedules processes for execution, allocating CPU time and managing process priorities.
11. **Kernel Module Support:** It allows the addition of dynamically loadable modules to extend kernel functionality without recompiling the entire kernel.
12. **System Initialization:** The kernel initializes the system during boot, setting up hardware, configuring devices, and launching system services.
13. **Error Handling:** It manages system errors, logging, and provides information about system status and issues.

These functions collectively make the kernel the heart of the Unix operating system, responsible for the proper functioning and resource management of the system.

#### 9. Which Process is Responsible for Getty Process.

getty is invoked by init. It is the second process in the series init-getty-login-shell, which ultimately connects a user with the Linux system. getty reads the user's login name and invokes the login command with the user's name as an argument.

#### 10. What is the use of Kill Command.

*kill* command in Linux (located in /bin/kill), is a built-in command which is used to terminate processes manually. *kill* command sends a signal to a process that terminates the process. If the user doesn't specify any signal which is to be sent along with the kill command, then a default *TERM* signal is sent that terminates the process.

##### Basic Syntax of `kill` command in Linux

The basic syntax of the `kill` command is as follows:

##### Syntax:

kill [signal] PID

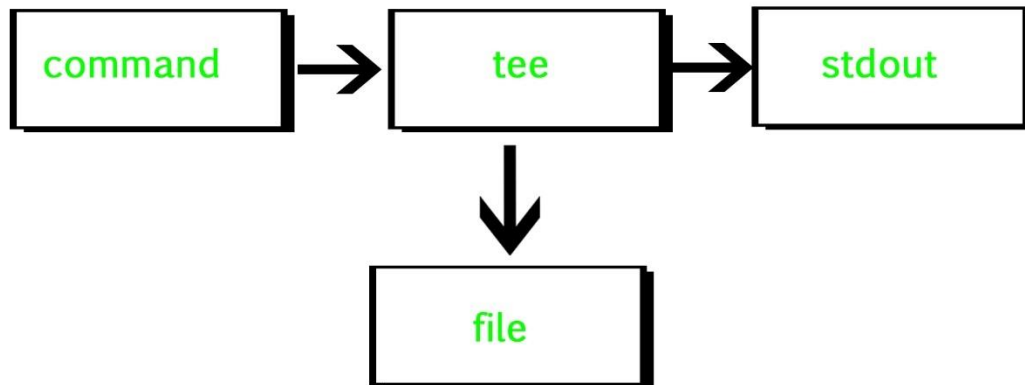
Here,

- **PID** = The `kill` command requires the process ID (PID) of the process we want to terminate.
- **[signal]** = We have to specify the signal and if we don't specify the signal, the default signal **TERM** is sent to terminate the process.



## 11. Explain Tee Command

**Tee command** reads the standard input and writes it to both the standard output and one or more files. The command is named after the T-splitter used in plumbing. It basically breaks the output of a program so that it can be both displayed and saved in a file. It does both the tasks simultaneously, copies the result into the specified files or variables and also display the result.



### SYNTAX:

**tee [OPTION]... [FILE]...**

#### Options :

**1.-a Option :** It basically do not overwrite the file but append to the given file.

Suppose we have **file1.txt**

Input: geek

for

geeks

and **file2.txt**

Input:geeks

for

geeks

### SYNTAX :

**geek@HP:~\$ wc -l file1.txt|tee -a file2.txt**

### OUTPUT :

3 file1.txt

geek@HP:~\$cat file2.txt

### OUTPUT:

geeks

for

geeks

3 file1.txt

```
anurag@HP: ~
geekforgeeks@~$:cat file1.txt
geek
for
geeks
geekforgeeks@~$:cat file2.txt
geek
for
geeks

geekforgeeks@~$:wc -l file1.txt|tee -a file2.txt
3 file1.txt
geekforgeeks@~$:cat file2.txt
geek
for
geeks

3 file1.txt
geekforgeeks@~$:
```

**2.-help Option :** It gives the help message and exit.

**SYNTAX :**

geek@HP:~\$ tee --help

```
anurag@HP: ~
anurag@HP:~$ tee --help
Usage: tee [OPTION]... [FILE]...
Copy standard input to each FILE, and also to standard output.

-a, --append                append to the given FILEs, do not overwrite
-i, --ignore-interrupts    ignore interrupt signals
-p                          diagnose errors writing to non pipes
--output-error[=MODE]      set behavior on write error.  See MODE below
--help                     display this help and exit
--version                  output version information and exit

MODE determines behavior with write errors on the outputs:
'warn'                     diagnose errors writing to any output
'warn-nopipe'              diagnose errors writing to any output not a pipe
'exit'                     exit on error writing to any output
'exit-nopipe'              exit on error writing to any output not a pipe
The default MODE for the -p option is 'warn-nopipe'.
The default operation when --output-error is not specified, is to
exit immediately on error writing to a pipe, and diagnose errors
writing to non pipe outputs.

GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
Full documentation at: <http://www.gnu.org/software/coreutils/tee>
or available locally via: info '(coreutils) tee invocation'
anurag@HP:~$
```

**3.-version Option :** It gives the version information and exit.

**SYNTAX :**

geek@HP:~\$ tee --version

```

anurag@HP: ~
anurag@HP:~$ tee --version
tee (GNU coreutils) 8.26
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Mike Parker, Richard M. Stallman, and David MacKenzie.
anurag@HP:~$

```

### Application

Suppose we want to count number of characters in our file and also want to save the output to new text file so to do both activities at same time, we use tee command.

```
geek@HP:~$ wc -l file1.txt | tee file2.txt
```

OUTPUT:

```
geek@HP:~$15 file1.txt
```

Here we have file1 with 15 characters, so the output will be 15 and the output will be stored to file2. In order to check the output we use :

```
geek@HP:~$ cat file2.txt
```

OUTPUT:

```
geek@HP:~$15 file1.txt
```

```

anurag@HP: ~
anurag@HP:~$ cat file1.txt
geek
for
geeks
anurag@HP:~$ wc -c file1.txt | tee file2.txt
15 file1.txt
anurag@HP:~$ cat file2.txt
15 file1.txt
anurag@HP:~$

```

## 12. Explain Internal and External Command in unix.

In Unix-like operating systems, "commands" refer to the instructions or programs that you can run from the command line to perform various tasks. These commands can be categorized into two main types: Internal and External commands.

1. **Internal Commands:** Internal commands are built into the shell (the command-line interface) itself. They are also known as built-in commands. These commands are part of the shell's functionality, and they are executed directly by the shell without the need to run an external program or script. Some common internal commands in Unix-like systems include:

- **cd**: Used to change the current working directory.
- **echo**: Displays a message or variable value to the terminal.
- **pwd**: Prints the current working directory.
- **exit**: Terminates the shell or a script.
- **alias**: Creates aliases for longer command sequences.
- **history**: Displays a list of previously executed commands.

Internal commands are generally faster to execute than external commands because they don't involve launching a separate program.

2. **External Commands**: External commands are standalone programs or scripts that are separate from the shell. When you run an external command, the shell locates the command in the system's directories (usually specified in the PATH environment variable) and executes it as a separate process. These commands are not built into the shell and are independent programs or scripts.

Examples of external commands are:

- **ls**: Lists files and directories in the current directory.
- **grep**: Searches for patterns in text files.
- **cp**: Copies files and directories.
- **rm**: Removes (deletes) files and directories.
- **find**: Searches for files and directories in a directory hierarchy.

External commands can be installed, updated, or removed independently from the shell, making them versatile and expandable.

### 13. Special File Permissions (setuid, setgid and Sticky Bit)

Three special types of permissions are available for executable files and public directories: setuid, setgid, and sticky bit. When these permissions are set, any user who runs that executable file assumes the ID of the owner (or group) of the executable file.

You must be extremely careful when you set special permissions, because special permissions constitute a security risk. For example, a user can gain superuser capabilities by executing a program that sets the user ID (UID) to 0, which is the UID of root. Also, all users can set special permissions for files that they own, which constitutes another security concern.

You should monitor your system for any unauthorized use of the setuid permission and the setgid permission to gain superuser capabilities. A suspicious permission grants ownership of an administrative program to a user rather than to root or bin. To search for and list all files that use this special permission.

## setuid Permission

When setuid permission is set on an executable file, a process that runs this file is granted access on the basis of the owner of the file. The access is **not** based on the user who is running the executable file. This special permission allows a user to access files and directories that are normally available only to the owner.

For example, the setuid permission on the passwd command makes it possible for users to change passwords. A passwd command with setuid permission would resemble the following:

```
-r-sr-sr-x 3 root sys 28144 Jun 17 12:02 /usr/bin/passwd
```

This special permission presents a security risk. Some determined users can find a way to maintain the permissions that are granted to them by the setuid process even after the process has finished executing.

---

### Note –

The use of setuid permissions with the reserved UIDs (0–100) from a program might not set the effective UID correctly. Use a shell script, or avoid using the reserved UIDs with setuid permissions.

---

## setgid Permission

The setgid permission is similar to the setuid permission. The process's effective group ID (GID) is changed to the group that owns the file, and a user is granted access based on the permissions that are granted to that group. The /usr/bin/mail command has setgid permissions:

```
-r-x--s--x 1 root mail 67504 Jun 17 12:01 /usr/bin/mail
```

When the setgid permission is applied to a directory, files that were created in this directory belong to the group to which the directory belongs. The files do not belong to the group to which the creating process belongs. Any user who has write and execute permissions in the directory can create a file there. However, the file belongs to the group that owns the directory, not to the group that the user belongs to.

You should monitor your system for any unauthorized use of the setgid permission to gain superuser capabilities. A suspicious permission grants group access to such a program to an

unusual group rather than to root or bin. To search for and list all files that use this permission

## Sticky Bit

The **sticky bit** is a permission bit that protects the files within a directory. If the directory has the sticky bit set, a file can be deleted only by the file owner, the directory owner, or by a privileged user. The root user and the Primary Administrator role are examples of privileged users. The sticky bit prevents a user from deleting other users' files from public directories such as /tmp:

```
drwxrwxrwt 7 root sys 400 Sep 3 13:37 tmp
```

## 14. Explain Timestamp in Unix.

Timestamps are records for the times in which actions are performed on files. A timestamp is useful because it keeps records of when a file was accessed, modified, or added. Linux's files have 3 timestamps recorded by the computer:

- **Access timestamp (atime):** which indicates the last time a file was accessed.
- **Modified timestamp (mtime):** which is the last time a file's contents were modified.
- **Change timestamp (ctime):** which refers to the last time some metadata related to the file was changed.

In Linux, a timestamp is actually stored as a number of seconds instead of a date and time. This number of seconds refers to the amount of time since 00:00:00 on January 1, 1970, which is the time of Unix Epoch. However, when a user wants a timestamp to be displayed, Linux will translate it to a human-readable format, so it is displayed as a date and time. Users can view timestamps using ls command or stat command.

### mtime:

Modified timestamp (mtime) indicates the last time the contents of a file were modified. For example, if new contents were added, deleted, or replaced in a file, the modified timestamp is changed. To view the modified timestamp, we can simply use the ls command with -l option.

Syntax:

```
ls -l [filename]
```

### ctime:

Unlike mtime, which is only related to the contents inside a file, changed timestamp indicates the last time some metadata of a file was changed. ctime refers to the last time when a file's metadata.

For example, if permission settings of a file were modified, ctime will indicate it. To see the changed timestamp, we can use -lc option with the ls command as follows:

Syntax:

```
ls -lc [filename]
```

### **atime:**

Access timestamp (atime) refers to the last time a file was read by a user. That is, a user displayed the contents of a file using any suitable program, but did not necessarily modify anything. To view an access timestamp using ls command, we use -lu option followed by the file name.

Syntax:

```
ls -lu [filename]
```

stat command:

stat command can be used to see all timestamps of a file simultaneously.

Syntax:

```
stat [filename]
```

### Comparison Table

The table below summarizes the difference between the three timestamps we mentioned:

	<b>File Contents are Modified</b>	<b>Metadata is Modified</b>	<b>File Accessed without Modification</b>	<b>Command to Use</b>
<b>mtime</b>	Changes	No change	No change	<i>ls -l</i> or <i>stat</i>
<b>ctime</b>	Changes	Changes	No change	<i>ls -cl</i> or <i>stat</i>
<b>atime</b>	Changes	No change	Changes	<i>ls -ul</i> or <i>stat</i>

To further explain the concept, we will examine a file named test.txt, the following changes were made to the file:

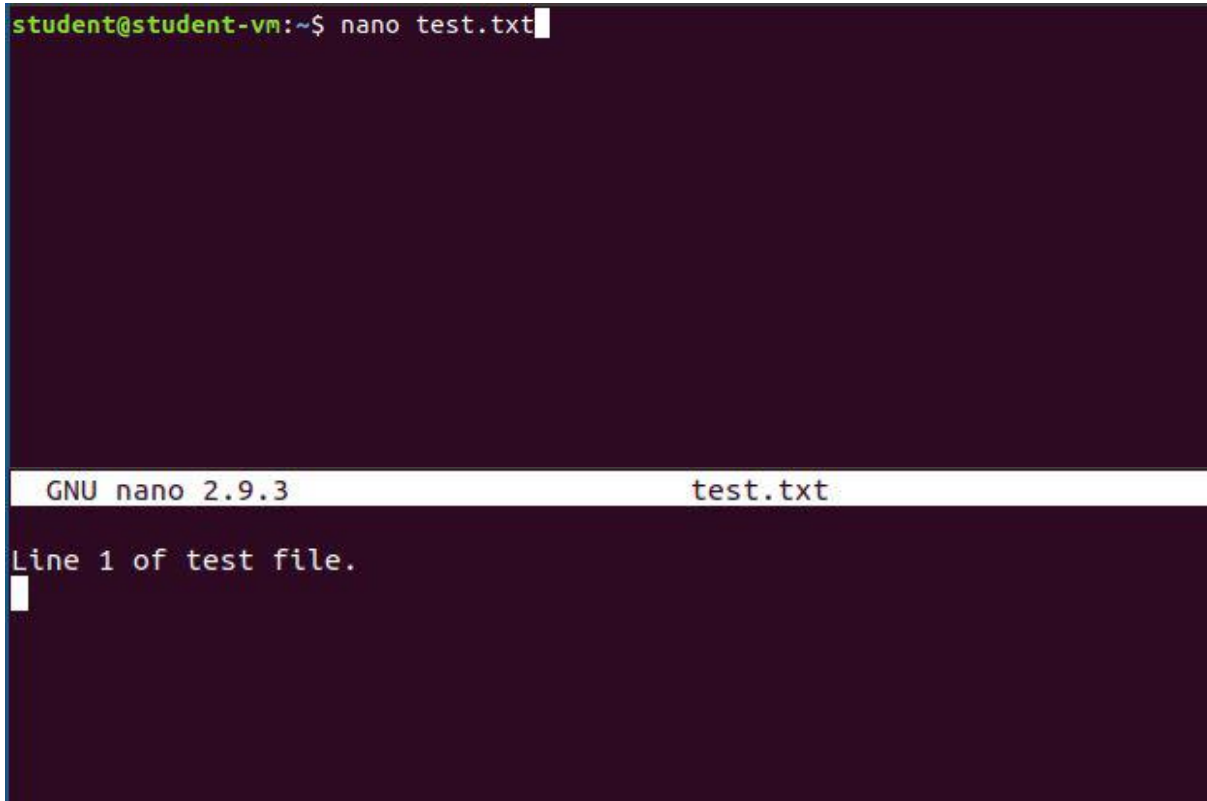
## Creating the File

The file was created at 14:04 on 25/03/2021 using the nano command. We can also use the touch command or any text editor. Initially, we added 1 line to the file

### Command to create file:

nano test.txt

```
student@student-vm:~$ nano test.txt
```



```
GNU nano 2.9.3 test.txt
Line 1 of test file.
```

Initially, the timestamps all show the time in which the file was created. The image below shows an example of using the stat command to view the 3 timestamps. In this image, the initial timestamps, which show the time that the file was created, are shown. The number +004 at the right of each timestamp is known as time zone offset, which indicates that the time zone is +004 hours ahead of UTC. The displayed date and time are converted from UTC to the local time zone when displayed to the user. We can also notice that stat command is highly exact in showing the seconds in a timestamp.

### Command:

stat test.txt

```
student@student-vm:~$ stat test.txt
  File: test.txt
  Size: 10          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d Inode: 266584       Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/  student)   Gid: ( 1000/  student)
Access: 2021-03-25 14:04:31.651010526 +0400
Modify: 2021-03-25 14:04:31.651010526 +0400
Change: 2021-03-25 14:04:31.651010526 +0400
 Birth: -
```

Alternatively, ls command can be used to view each timestamp individually as follows:

### Command:



mtime: ls -l test.txt

ctime: ls -cl test.txt

atime: ls -ul test.txt

```
student@student-vm:~$ ls -l test.txt
-rw-r--r-- 1 student student 10 Mar 25 14:04 test.txt
student@student-vm:~$ ls -cl test.txt
-rw-r--r-- 1 student student 10 Mar 25 14:04 test.txt
student@student-vm:~$ ls -ul test.txt
-rw-r--r-- 1 student student 10 Mar 25 14:04 test.txt
student@student-vm:~$
```

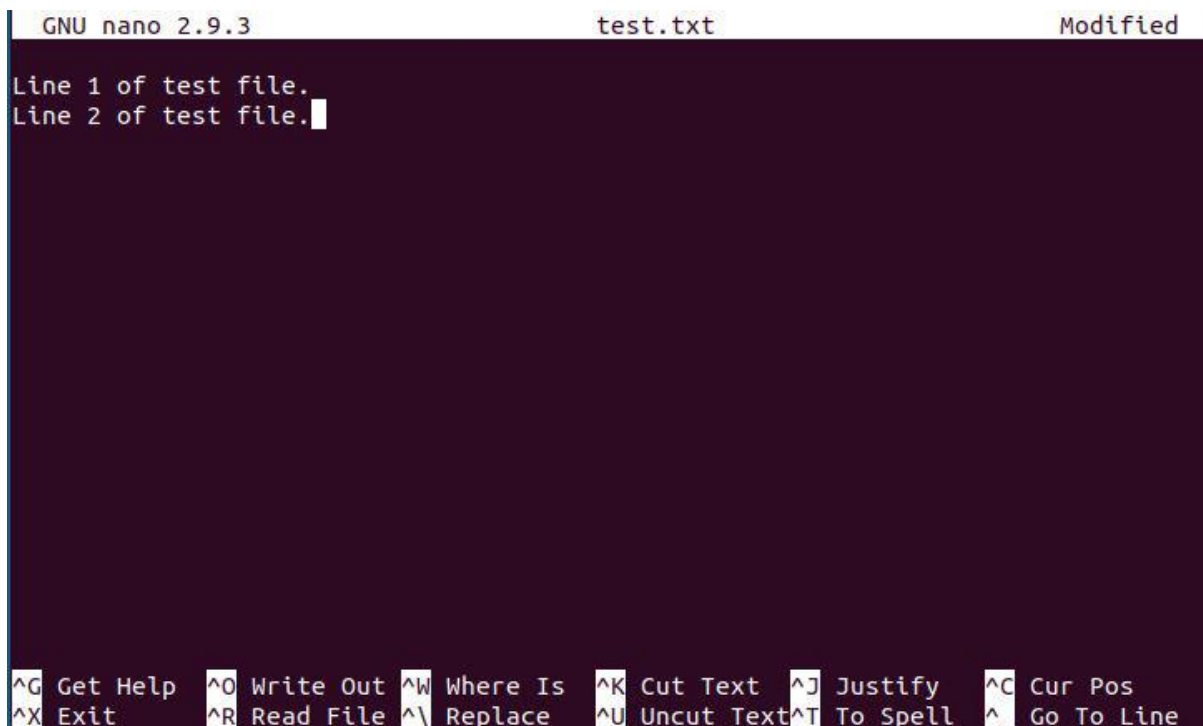
In the following steps, we will make some changes in the file and observe the change in timestamps using stat command. ls command can be used as well.

### Modifying the File

The file was accessed and a new line was added to it at 14:22 on 25/03/2021 using nano text editor (any text editor can be used).

#### Command:

nano test.txt



```
GNU nano 2.9.3 test.txt Modified
Line 1 of test file.
Line 2 of test file.
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
```

Using stat command, we can see that all 3 timestamps were changed to 14:22.

#### Command:

stat test.txt

```
student@student-vm:~$ stat test.txt
  File: test.txt
  Size: 17          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d Inode: 266584       Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/ student)   Gid: ( 1000/ student)
Access: 2021-03-25 14:22:47.903030220 +0400
Modify: 2021-03-25 14:22:53.399030319 +0400
Change: 2021-03-25 14:22:53.399030319 +0400
 Birth: -
student@student-vm:~$
```

## Changing Metadata

The file's permissions were changed at 14:36 on 25/3/2021 using chmod command.

### Command:

chmod 777 test.txt

```
student@student-vm:~$ chmod 777 test.txt
student@student-vm:~$
```

We notice that after changing the permissions, both ctime and atime changed to 14:36. This is not the case with mtime since it only changes when the contents inside the file are modified. Therefore, mtime is still at 14:22.

### Command:

stat test.txt

```
student@student-vm:~$ stat test.txt
  File: test.txt
  Size: 17          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d Inode: 266584       Links: 1
Access: (0777/-rwxrwxrwx)  Uid: ( 1000/ student)   Gid: ( 1000/ student)
Access: 2021-03-25 14:22:47.903030220 +0400
Modify: 2021-03-25 14:22:53.399030319 +0400
Change: 2021-03-25 14:36:50.239045352 +0400
 Birth: -
student@student-vm:~$
```

## Opening the File without Making Changes

The file was opened in nano text editor, but no changes were made at 14:55 on 25/3/2021.

### Command:

nano test.txt

```
GNU nano 2.9.3 test.txt
Line 1 of test file.
Line 2 of test file.

[ Read 2 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
```

From the output of stat command, we can observe that the only timestamp that changed to 14:55 is the access timestamp. This is because no data was changed. Therefore, ctime remains at 14:36 while mtime remains at 14:22.

### Command:

stat test.txt

```
student@student-vm:~$ stat test.txt
  File: test.txt
  Size: 17          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d Inode: 266584       Links: 1
Access: (0777/-rwxrwxrwx)  Uid: ( 1000/ student)   Gid: ( 1000/ student)
Access: 2021-03-25 14:55:05.707065032 +0400
Modify: 2021-03-25 14:22:53.399030319 +0400
Change: 2021-03-25 14:36:50.239045352 +0400
 Birth: -
student@student-vm:~$
```